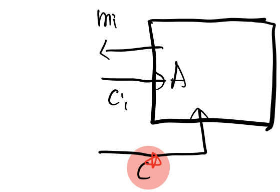
Authenticated Encryption and KDF

인증암호 키 유도함수

Jong Hwan Park

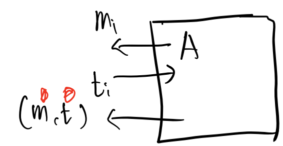
Recap

- **Confidentiality:** semantic security against a CPA attack
  - 메시지를 숨기기
  - Encryption secure against ~~eavesdropping only~~ CPA
  - 강력하다.
  - CBC, CTR
  - 숨기기



암호문자체 변경X

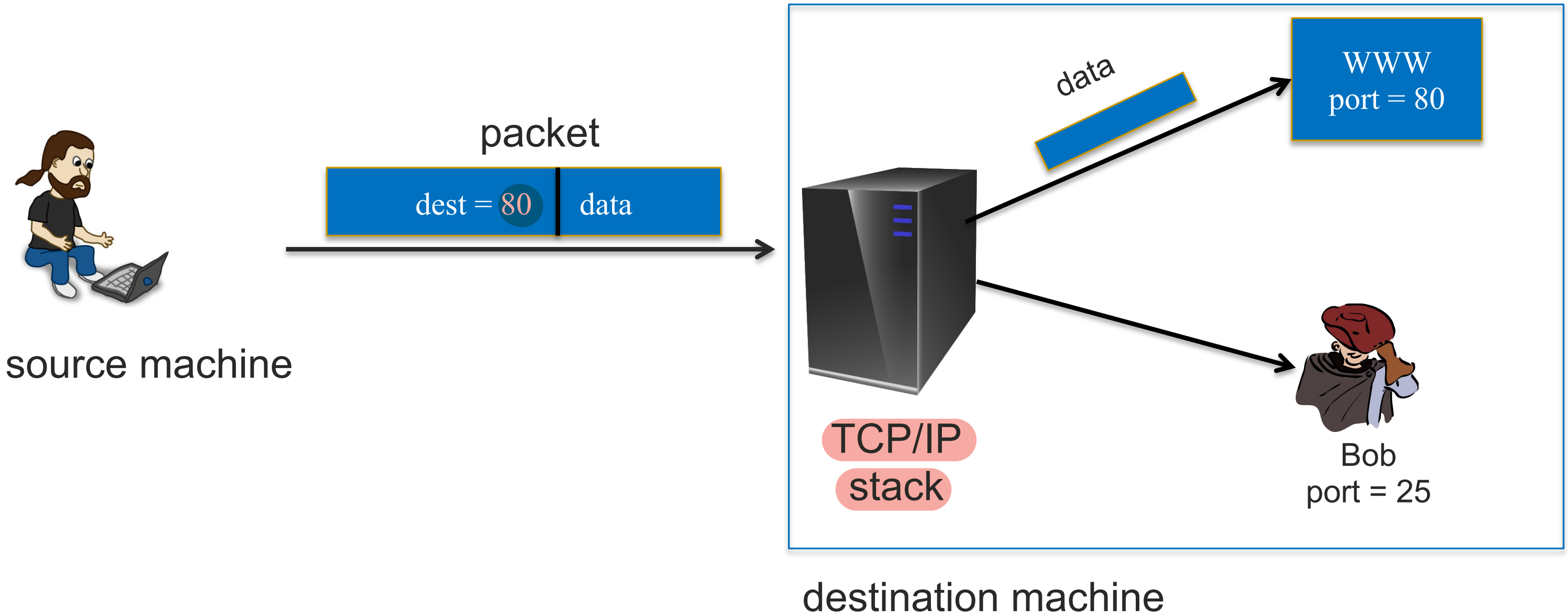
- **Integrity:** unforgeability against a CMA attack
  - 무결성
  - Plus sender authentication
  - CBC-MAC (or CMAC), HMAC
  - 위조, 변조 방지.



- What about active adversaries that can inject, modify, …
  - Encryption against tampering (원문과 암호문 변경)
  - Need encryption that ensures both confidentiality and integrity 둘다 대칭키 (k_e, k_m)
  - How to securely combine CPA-secure encryption and secure MAC
  - CMA-Secure

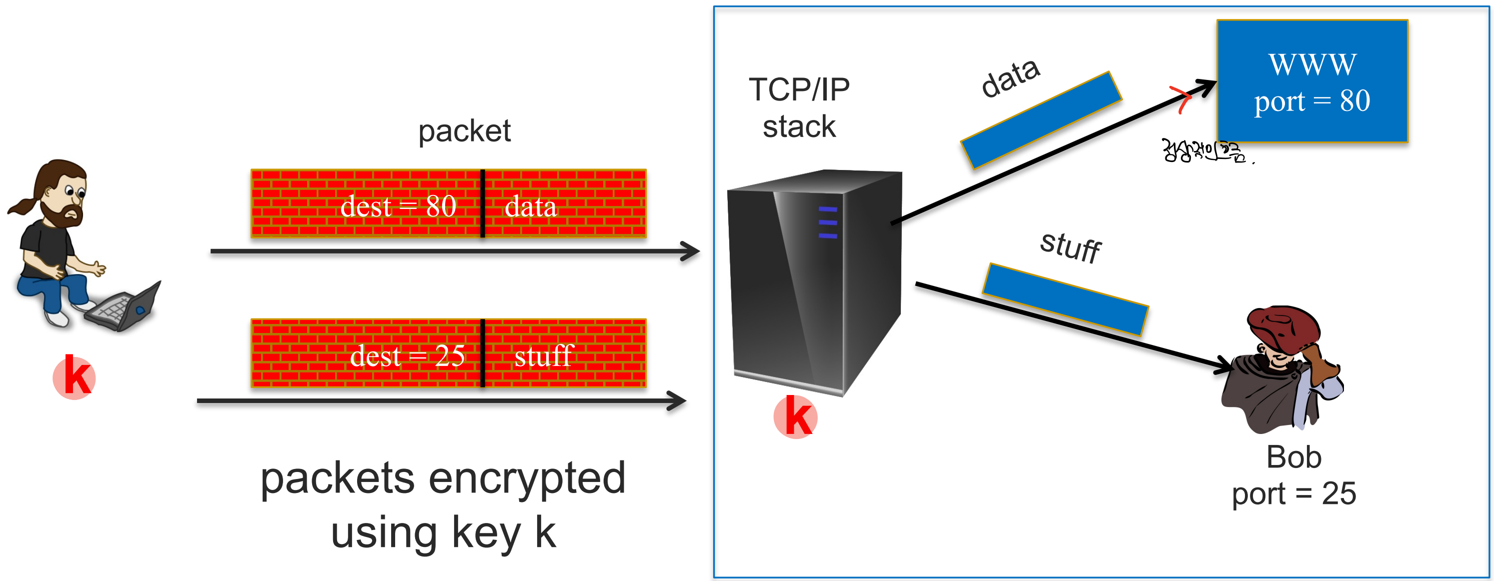
Sample Tampering Attacks (1)

TCP/IP: (highly abstracted)



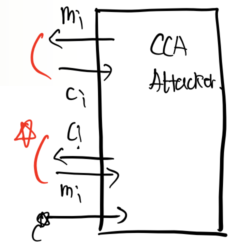
Sample Tampering Attacks (2)

IPsec: (highly abstracted)

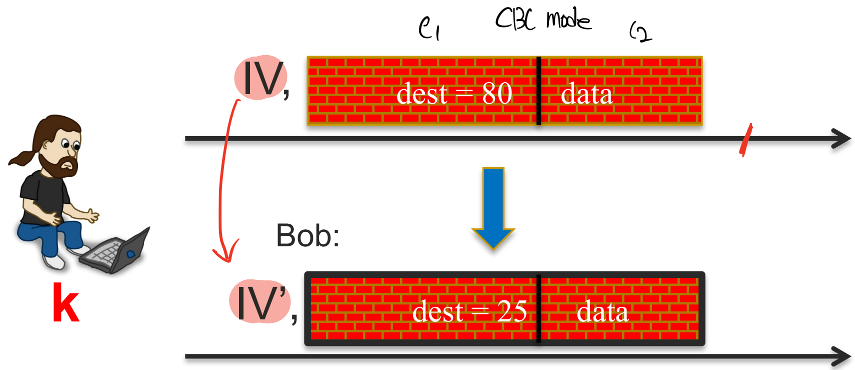


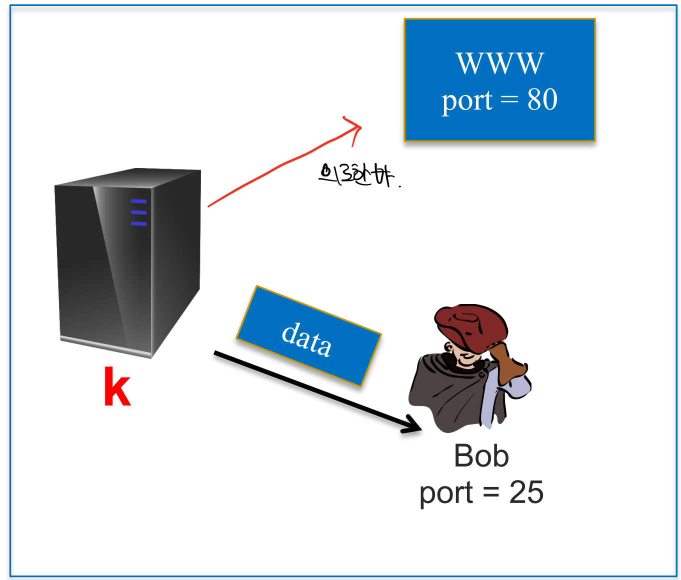
Sample Tampering Attacks (3)

CCA공격



Note: attacker obtains decryption of any ciphertext beginning with “dest=25”





Easy to do for CBC with rand. IV

(only IV is changed)

→ What should IV’ be?

$$m[0] = D(k, c[0]) \oplus IV = \text{“dest=}\underline{80}\text{…”}$$

25

00

$$IV' = IV \oplus (\ldots\underline{80}\ldots) \oplus (\ldots\underline{25}\ldots)$$

Authenticated Encryption

An **authenticated encryption** system (E,D) is a cipher where

As usual: $E:\ K \times M \times N \longrightarrow C$ Enc(K,M) → C

but $D:\ K \times C \times N \longrightarrow M \cup \{\perp\}$ Dec(K,C) → M ∪ {⊥}

error

=암호문이 원문그대로 유지X

Nonce 살리면

Nonce based AE

Invalid ciphertext is rejected

Security: the system must provide

- **confidentiality** under a CPA attack, and
- **ciphertext integrity:**
  - attacker cannot create new ciphertexts that decrypt properly

---

Note: CBC with random IV, CTR with random IV does not provide ciphertext integrity
- $D(k, \cdot)$ never outputs $\perp$

AE Construction… but first, some history

Authenticated Encryption (AE): introduced in 2000 [KY'00, BN'00]

Crypto APIs before then: (e.g. MS-CAPI) crypto API

- Provide API for CPA-secure encryption (e.g. CBC with rand. IV)
- Provide API for MAC (e.g. HMAC)

Every project had to combine the two itself without a well defined goal

- Not all combinations provide AE …

AE Construction from Enc and MAC

Let (E,D) be CPA-secure cipher and (S,V) secure MAC

Encryption key k_E. MAC key = k_I $= k_m$

Option 1: (SSL) **Mac-then-encrypt** ex) Https

msg m ⇨ $S(k_I, m)$: msg m | tag ⇨ $E(k_E, m \| tag)$

Option 2: (IPsec) **Encrypt-then-Mac** ex) IpSec

always correct

msg m ⇨ $E(k_E, m)$ ⇨ $S(k_I, c)$: tag

Option 3: (SSH) **Encrypt-and-Mac** ex) 서버에 접속시 SSH

msg m ⇨ $E(k_E, m)$ ⇨ $S(k_I, m)$: tag

Standards (at a high level)

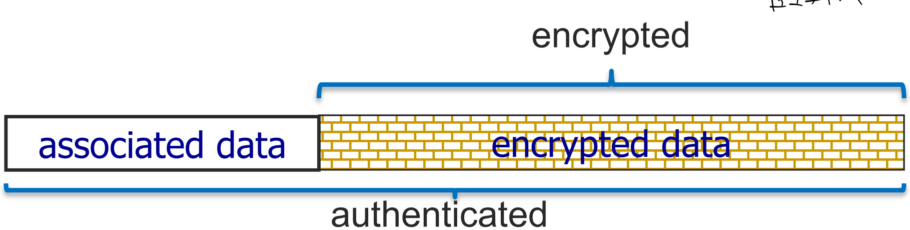
- **GCM**: CTR mode encryption then CW-MAC
  (accelerated via Intel's PCLMULQDQ instruction)
- **CCM**: CBC-MAC then CTR mode encryption (802.11i)
- **EAX**: CTR mode encryption then CMAC

- All are nonce-based

대칭 키 암호의 최종형태

- In practice, all support AEAD: (AE with Associated Data)

= 암호화 X



AEAD

- An **AEAD** system (E, D) is a cipher where

Nonce-based

$$E_K(N, A, M) \longrightarrow C$$ (4개의 input)

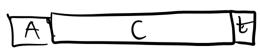
$$D_K(N, A, C) \longrightarrow M \cup \{\perp\}$$

- How does CCM encryption work?
- How does EAX encryption work?
- How does GCM encryption work?
- …

- Inputs:
  - N : nonce = number used *at once* (e.g., sequence number, counter)
  - A : associated data, not encrypted (e.g., IP address)
  - M : message
  - C : ciphertext that consists of (CT, tag)

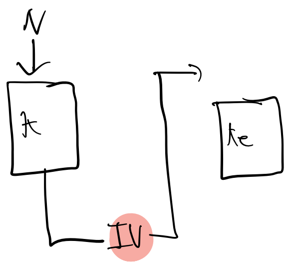
Encrypt-then-MAC in AEAD

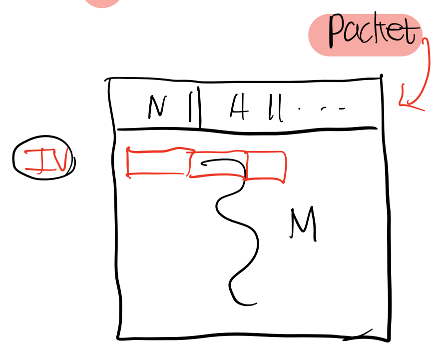
- GCM (or EAX) mode
  - Assume Alice and Bob share the symmetric key $K=(K_e, K_m)$
  - Two algorithms E=(Enc, Dec) and MAC=(Mac, Verify) are used
  - Encrypt(K, N, A, M)
    - $C=\text{Enc}(K_e, M)$
    - $T=\text{MAC}(K_m, N\|A\|C)$
    - Output (C, T)



  - Decrypt(K, N, A, (C,T))
    - $\text{Verify}(K_m, N\|A\|C, T)=1/0$
    - If T is invalid, output ⊥
    - Otherwise, output $\text{Dec}(K_e, C)=M$

Nonce-based CCBC, CTR)





MAC-then-encrypt in AEAD

- CCM mode
  - Assume Alice and Bob share the symmetric key $K=(K_e, K_m)$
  - Two algorithms E=(Enc, Dec) and MAC=(Mac, Verify) are used

  - Encrypt(K, N, A, M)
    - $T=MAC(K_m, N||A||M)$
    - $C=Enc(K_c, M||T)$
    - Output C

  - Decrypt(K, N, A, C)
    - $Dec(K_e, C)=M||T$
    - $Verify(K_m, N||A||M, T)=1/0$
    - If T is invalid, output $\perp$
    - Otherwise, output M

Encrypt-and-MAC in AEAD

- Encrypt-and-MAC mode SSH
  - Assume Alice and Bob share the symmetric key $K=(K_e, K_m)$
  - Two algorithms E=(Enc, Dec) and MAC=(Mac, Verify) are used

  - Encrypt(K, N, A, M)
    - $C=Enc(K_e, M)$
    - $T=MAC(K_m, N||A||M)$
    - Output (C, T)

  - Decrypt(K, N, A, (C,T))
    - $Dec(K_e, C)=M$
    - $Verify(K_m, N||A||M, T)=1/0$
    - If T is invalid, output ⊥
    - Otherwise, output M

An Example API (ccm mode)

```
int tc_ccm_generation_encryption(uint8_t *out, unsigned int olen,
                                 const uint8_t *associated_data,
                                 unsigned int alen, const uint8_t *payload,
                                 unsigned int plen, TCCcmMode_t c)

int tc_ccm_decryption_verification(uint8_t *out, unsigned int olen,
                                   const uint8_t *associated_data,
                                   unsigned int alen, const uint8_t *payload,
                                   unsigned int plen, TCCcmMode_t c)
```

AES-CCM
기반

K, N, A, M, C

- Homework 5

- https://github.com/intel/tinycrypt
    - AES-CCM mode code analysis (due date: 11/04)

Performance:

AMD Opteron, 2.2 GHz (Linux) Crypto++ 5.6.0 [Wei Dai]

Cipher	code size	Speed (MB/sec)		
AES/GCM	large **	108	AES/CTR	139
AES/CCM	smaller	61	AES/CBC	109
AES/EAX	smaller	61		
			AES/CMAC	109
			HMAC/SHA1	147

* extrapolated from Ted Kravitz's results ** non-Intel machines

Case Study: TLS Record Protocol (TLS 1.2)

$k_{b \to s}$, $k_{s \to b}$ $k_{b \to s}$, $k_{s \to b}$

송신 수신 키가 다르다.

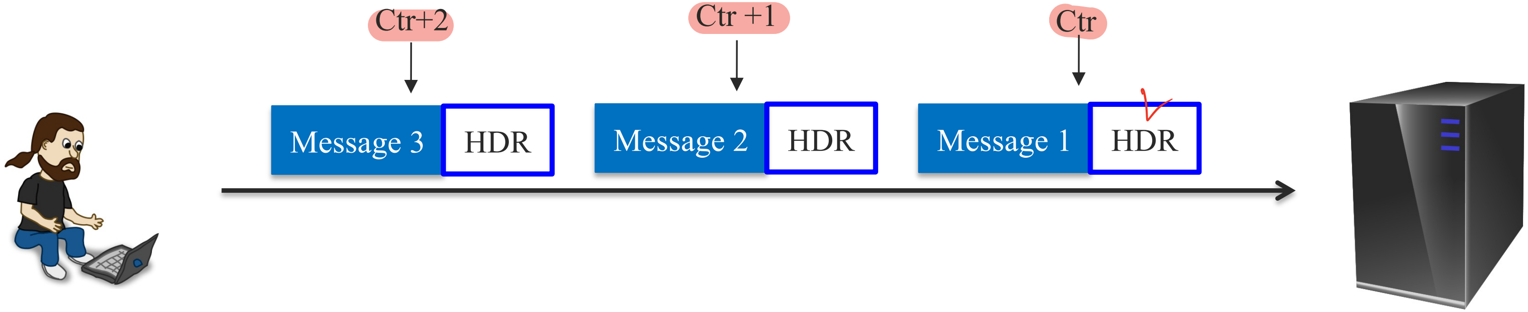
Unidirectional keys: $k_{b \to s}$ and $k_{s \to b}$

Stateful encryption:

- Each side maintains two 64-bit counters: $ctr_{b \to s}$, $ctr_{s \to b}$
- Init. to 0 when session started. ctr++ for every record.
- Purpose: replay defense =일종의 sequence number.

Case Study: TLS Record Protocol (TLS 1.2)

MAC then encrypt



$K=(k_{enc}, k_{mac})$ $K=(k_{enc}, k_{mac})$

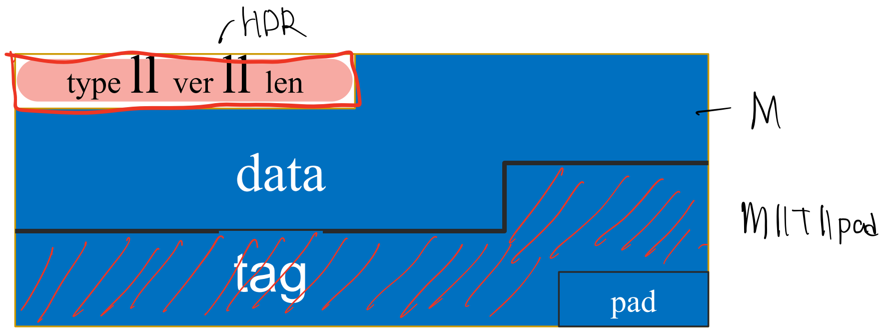
Unidirectional keys: $k_{b \to s}$ and $k_{s \to b}$

Stateful encryption:

- Each side maintains two 64-bit counters: $ctr_{b \to s}$, $ctr_{s \to b}$
- Init. to 0 when session started. ctr++ for every record.
- Purpose: replay defense

TLS record: encryption (CBC AES-128, HMAC-SHA1)

$k_{b\to s} = (k_{mac}\ ,\ k_{enc})$



Browser side **enc($k_{b\to s}$, data, $ctr_{b\to s}$) :**

not transmitted in packet

step 1: $\quad tag \leftarrow MAC\big(k_{mac}\ ,\ [\ \overset{N}{++ctr_{b\to s}}\ \text{ll}\ \overset{A}{\text{header}}\ \text{ll}\ \overset{M}{\text{data}}\ ]\big)$

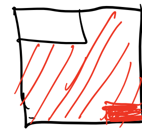
step 2: pad [header ll data ll tag] to AES block size

step 3: CBC encrypt with k_{enc} and new random IV

(nonce-based

step 4: prepend header

TLS record: decryption (CBC AES-128, HMAC-SHA1)



Server side **dec($k_{b\to s}$, record, $ctr_{b\to s}$) :**

step 1: CBC decrypt record using k_{enc}

step 2: check pad format: send bad_record_mac if invalid

step 3: check tag on $[\ {+}{+}ctr_{b\to s}^{\ N}$ ll header^{A} ll data^{M}]

send bad_record_mac if invalid

완주후 error

→ Two types of error: **padding error / MAC error** M 아 ⊥

Provides authenticated encryption

(provided no other info. is leaked during decryption)

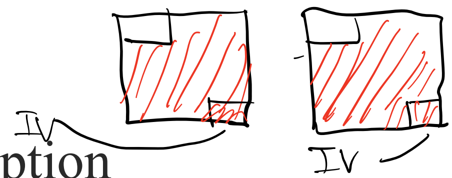
Nonce-based AEAD

Bugs in Older Versions (prior to TLS 1.1)

- **IV for CBC is predictable:** (chained IV)

  IV for next record is last ciphertext block of current record.

  Not CPA secure



- **Padding oracle**: during decryption

  if pad is invalid send decryption failed alert $\perp$

  if mac is invalid send bad_record_mac alert $\perp$

$\Rightarrow$ attacker learns info. about plaintext (attack in next slides)

Lesson: when decryption fails, do not explain why

#256

$k = \overbrace{1100111110001101001\,10 \cdots 010 \cdots 011101}$

k_e k_m

누군가 랜덤처럼 분포X.

지금까지 보안적으로는 random하게 들어가야한다.

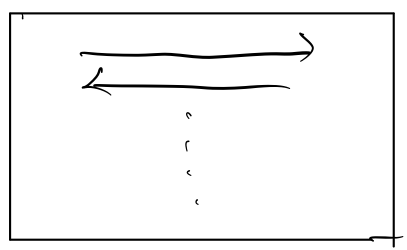
Key Derivation Function (KDF)

k — key exchange protocol — k

$\boxed{k_e \mid k_m}$

k = entropy는 충분

$KDF(k) \rightarrow (k_e, k_m)$



$KDF(k) \rightarrow (k_e, k_m)$

Deriving many keys from One

Typical scenario. a single source key (SK) is sampled from:

- Hardware RNG (may produce biased output)
- A key exchange protocol (key uniform in some subset of K)
- SK may not be uniformly random

Need many keys to secure session:

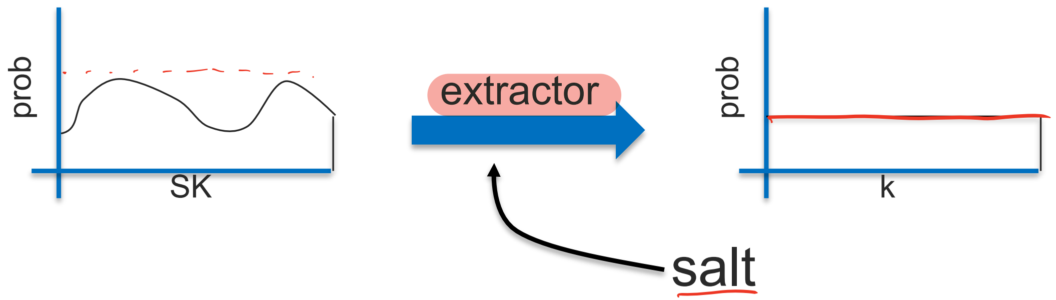
- Multiple keys for k_{enc}, k_{mac}, and others

Goal: generate many keys from this one source key

$$\boxed{SK} \xrightarrow[\text{KDF}]{} \boxed{k_1,\ k_2,\ k_3,\ \ldots}$$

Extract-then-Expand Paradigm

Step 1: **extract** pseudo-random key k from source key SK



salt: a non-secret string chosen at random

Step 2: **expand** k by using it as a PRF key

HKDF: PRF=HMAC

Standardized method

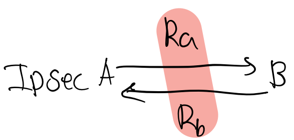
Implements the extract-then-expand paradigm:

uniform한 분포

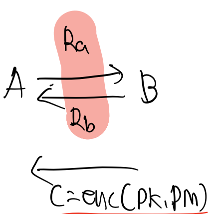
- Extract: using **k** ← **HMAC(salt, SK)**

HMAC key   HMAC data

$R_a \| R_b$ = Salt



  - Salt: a non-secret string chosen at random
    - In IPSec, salt consists of ($R_a \| R_b$) and k ← **HMAC**($R_a \| R_b$, g^{xy})
    - In SSL/TLS, salt consists of PM and k ← **HMAC**(**PM**, $R_a \| R_b$)



- Expand: using HMAC as a PRF with key **k** (next)

Expand (using PRF)

F: a PRF with key space K and outputs in $\{0,1\}^n$

Suppose PRF key k is uniform in K

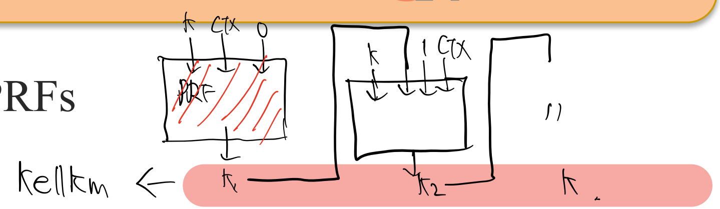
ex) CTX="PRF expansion"

- Define Expand as:

Expand(k, CTX, L) :=

$\mathrm{PRF}(k, (\mathbf{CTX}\|\mathbf{0})) \| \mathrm{PRF}(k, (K_1\|\mathbf{CTX}\|\mathbf{1})) \| \cdots \| \mathrm{PRF}(k, (K_{L-1}\|\mathbf{CTX}\|\mathbf{L}))$

(counter.



- **PRF:** HMAC or other constructed PRFs
- **Ki:** i-th previous PRF output
- **CTX:** a context string that uniquely identifies the application

  → even if two apps sample same SK they get indep. keys

PBKDF: Password-Based KDF

$H(pw) \to k$

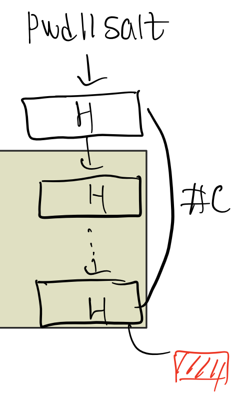
$\to sk$

Deriving keys from passwords:

- Do not use HKDF: passwords have insufficient entropy
- Derived keys will be vulnerable to dictionary attacks $2^{20} \leftarrow 2^{64}$

(more on this later)

PBKDF defenses: **salt** and **slow hash function**



Standard approach: **PKCS#5** (PBKDF1)

$H^{(c)}$(pwd ll salt): iterate hash function c times

iteration

random string

10^4

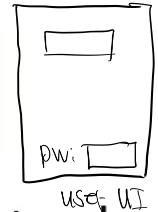
PBKDF2: enhanced in terms of (SHA-256, c times, salt-length)

Widely used: key storage, Wi-Fi access, PW-based encryption,...

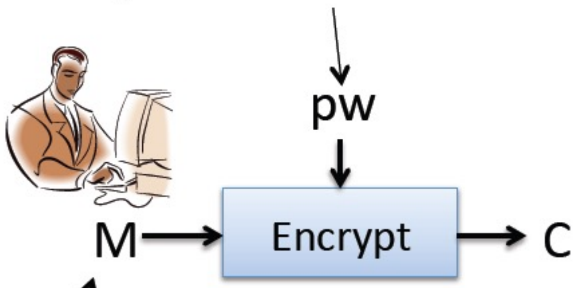
⇒ password를 넣어서 중요 데이터에 접근할때.

$PBKDF2^{(c)}(pw \,\|\, salt) = k$

Password-Based Encryption



secret ***password*** user remembers



Message
- English text
- RSA secret keys
- Password vaults
- ...

PKCS#5 is dominant standard

$PBKDF_2^{(c)}(pw \| salt) = k$

$c = CTR(k, m)$

H(pw) salt

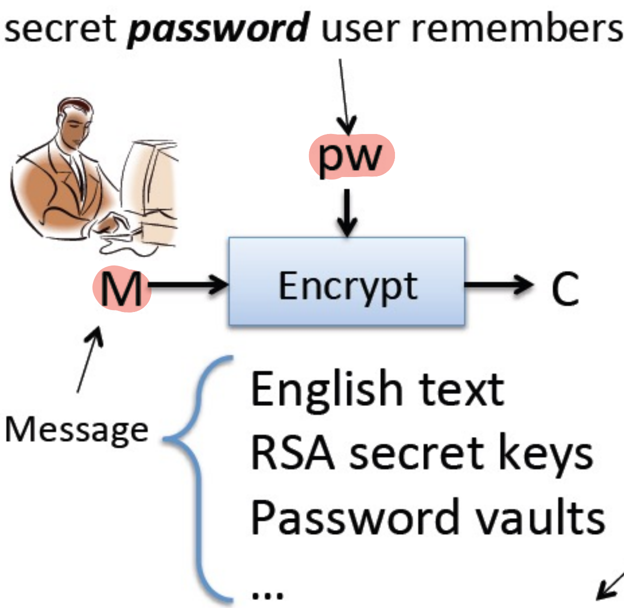
"PBE"

$PBKDF2^{(c)}(pw \| salt) = k$

$c = CBC(k, \text{signing key})$

signing key , salt

Password-Based Encryption

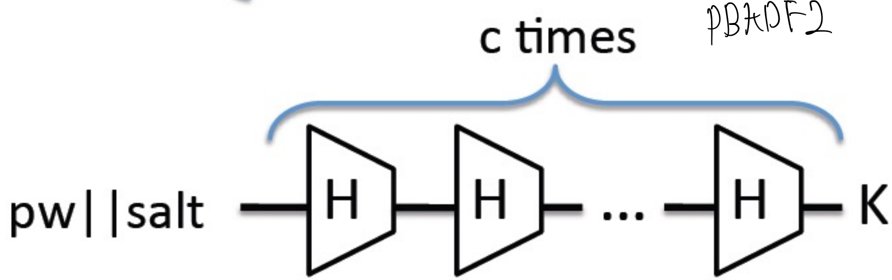


PKCS#5 is dominant standard

Encrypt(pw, M)
salt $\leftarrow_\$ \{0,1\}^{128}$
$K \leftarrow H^c(pw \,||\, salt)$
$C \leftarrow K \oplus M$
Return (salt,C)

Decrypt(pw, salt,C)
$K \leftarrow H^c(pw \,||\, salt)$
$M \leftarrow K \oplus C$
Return M

PBKDF2



Cryptographic hash function H
(H = SHA-256, SHA-512, etc.)

Common choice is c = 10,000

Q & A